APPENDIX

```
using System;
using System.Runtime.InteropServices;
using System.Security.Cryptography;
using System.Collections;

namespace ManagedI.Common
{
    #region Structs
    [Serializable]
    public enum KeySizes : ushort
    {
        SIZE384 = 0,
        SIZE512 = 1,
        SIZE1024 = 2,
        SIZE2048 = 3,
        SIZE4096 = 4
    }

    [Serializable]
    public enum CryptoTypes : ushort
    {
        RSA = 0
    }

    [Serializable]
    public enum CredsLevels : ushort
    {
        PrivateID = 0,
        Identity = 1,
        Domain = 2,
        DomainAndServer = 3,
        Email = 4,
        EmailAndServer = 5
    }

    [Serializable]
    public sealed class MICACredential
    {
        public KeySizes KeySize;
        public CryptoTypes CryptoType;
        public CredsLevels CredsLevel;

        public MICACredential()
        {
            KeySize = KeySizes.SIZE384;
            CryptoType = CryptoTypes.RSA;
            CredsLevel = CredsLevels.Email;
        }

        public override string ToString()
        {
            return CredentialString;
        }

        public string CredentialString
        {
            get
            {
                return ((ushort)KeySize).ToString() + ((ushort)CryptoType).ToString() + (↙
    (ushort)CredsLevel).ToString();
            }
            set
            {
                try
                {
                    if(value.Length < 3) throw(new Exception("Invalid license string was ↙
    supplied"));
                    KeySize = (KeySizes)Convert.ToUInt16(value.Substring(0, 1));
                    CryptoType = (CryptoTypes)Convert.ToUInt16(value.Substring(1, 1));
                    CredsLevel = (CredsLevels)Convert.ToUInt16(value.Substring(2, 1));
                }
                catch(Exception ex)
                {
                    System.Diagnostics.Debug.Assert(false, ex.Message);
                    throw new Exception("Credential String Error");
                }
            }
        }
    }

    [Serializable]
    public enum RenewalType : ushort
    {
        Automatic = 0,
        AskFirst = 1,
        AutoNotify = 2
    }

    [Serializable]
    public enum SharingType : ushort
    {
        Forbidden = 0,
        AskMe = 1,
        NotifyMe = 2,
        Allow = 3
    }

    [Serializable]
    public enum UnauthorizedUseType : ushort
    {
        Discard = 0,
        Challenge = 1,
        AskForwarder = 2,
        AskMe = 3,
        EnrollSilently = 4
    }

    [Serializable]
    public enum RequiredCryptoType : ushort
    {
        NONE = 0,
        SIZE384 = 1,
        SIZE1024 = 2,
        SIZE2048 = 3,
        SIZE4096 = 4
    }

    [Serializable]
    public sealed class MICALicense
    {
        public DateTime Expires;
        public RenewalType Renewal;
        public SharingType Sharing;
        public UnauthorizedUseType UnauthorizedUse;
        public RequiredCryptoType RequiredCrypto;
        public bool AutoSync;

        public MICALicense()
        {
            Expires = DateTime.MaxValue;
            Renewal = RenewalType.Automatic;
            Sharing = SharingType.Forbidden;
        UnauthorizedUse = UnauthorizedUseType.Discard;
        RequiredCrypto = RequiredCryptoType.NONE;
        AutoSync = false;
    }

    public override string ToString()
    {
        return LicenseString;
    }

    public string LicenseString
    {
        get
        {
            string license = Expires.ToString("yyMMdd");
            license += ((ushort)Renewal).ToString() + ((ushort)Sharing).ToString() + ↙
((ushort)UnauthorizedUse).ToString()
                + ((ushort)RequiredCrypto).ToString();
            if(AutoSync)
                license += "1";
            else
                license += "0";
            return license;
        }
        set
        {
            try
            {
                if(value.Length < 11) throw(new Exception("Invalid license string was↙
 supplied"));
                string date = value.Substring(0, 6);
                string renewal = value.Substring(6, 1);
                string sharing = value.Substring(7, 1);
                string unauthorizedUse = value.Substring(8, 1);
                string requiredCrypto = value.Substring(9, 1);
                string autoSync = value.Substring(10, 1);

                this.Expires = new DateTime(Convert.ToInt32(date.Substring(0, 2))
                    , Convert.ToInt32(date.Substring(2, 2)), Convert.ToInt32(date.    ↙
Substring(4, 2)));
                this.Renewal = (RenewalType)Convert.ToUInt16(renewal);
                this.Sharing = (SharingType)Convert.ToUInt16(sharing);
                this.UnauthorizedUse = (UnauthorizedUseType)Convert.ToUInt16(         ↙
unauthorizedUse);
                this.RequiredCrypto = (RequiredCryptoType)Convert.ToUInt16(           ↙
requiredCrypto);
                if(autoSync == "1")
                    this.AutoSync = true;
                else
                    this.AutoSync = false;
            }
            catch(Exception ex)
            {
                System.Diagnostics.Debug.Assert(false, ex.Message);
                throw new Exception("License String Error");
            }
        }
    }
}

#endregion Structs
[Serializable]
public class MICA
{
    #region Member Variables
    private RSACryptoServiceProvider rsa;
```

```
        private RSAParameters rsaparams;
        private Guid id;
        private SortedList additionalCreds;
        private string recipient;
        private string domain;
        private MICALicense license;
        private MICACredential credential;
        private string address;
        private string sigstring;
        private string keystring;
        private string credstring;
        #endregion Member Variables
        #region Constructors
        public MICA()
        {
            Reset();
            ResetKeys();
            this.id = Guid.NewGuid();
        }

        public MICA(Guid id)
        {
            Reset();
            ResetKeys();
            this.id = id;
        }

        public MICA(string recipient, string domain, MICALicense license, MICACredential ↙
    credential, SortedList additionalCreds)
        {
            Reset();
            Recipient = recipient;
            License = license;
            Credential = credential;
            AdditionalCreds = additionalCreds;
            Domain = domain;
            ResetKeys();
        }

        public MICA(string mica)
        {
            Reset();
            System.Text.UnicodeEncoding ue = new System.Text.UnicodeEncoding();

            // Start by seperating the parts
            string work = mica;
            work = work.Replace("_", "/");
            work = work.Replace("~", "+");

            int n = work.LastIndexOf("@");
            Domain = work.Substring(n + 1);
            work = work.Substring(0, n);

            n = work.LastIndexOf(".");
            Credential.CredentialString = work.Substring(n+1);
            work = work.Substring(0, n);

            n = work.LastIndexOf(".");
            License.LicenseString = work.Substring(n+1);
            work = work.Substring(0, n);

            ResetKeys();

            n = work.LastIndexOf(".");
            Signature = work.Substring(n+1);
            KeyString = work.Substring(0, n);
        }

        #endregion Constructors
        #region Private methods
        private void InitializeRSA()
        {
            switch(Credential.KeySize)
            {
                case KeySizes.SIZE384:
                    rsa = new RSACryptoServiceProvider(384);
                    break;
                case KeySizes.SIZE512:
                    rsa = new RSACryptoServiceProvider(512);
                    break;
                case KeySizes.SIZE1024:
                    rsa = new RSACryptoServiceProvider(1024);
                    break;
                case KeySizes.SIZE2048:
                    rsa = new RSACryptoServiceProvider(2048);
                    break;
                case KeySizes.SIZE4096:
                    rsa = new RSACryptoServiceProvider(4096);
                    break;
                default:
                    throw(new Exception("Invalid MICA Key Size specified for the MICA"));
            }
            rsaparams = rsa.ExportParameters(true);
            keystring = null;
            sigstring = null;
        }

        private void Reset()
        {
            id = Guid.Empty;
            address = null;
            this.AdditionalCreds = new SortedList();
            Recipient = null;
            Domain = null;
            sigstring = null;
            keystring = null;
            credstring = null;
            License = new MICALicense();
            Credential = new MICACredential();
        }

        private void ResetKeys()
        {
            switch(Credential.CryptoType)
            {
                case CryptoTypes.RSA:
                    InitializeRSA();
                    break;
                default:
                    System.Diagnostics.Debug.Assert(false, "Invalid Crypto Type specified↙
    for the MICA");
                    break;
            }
        }

        private string CredentialsString
        {
            get
            {
                if(credstring != null) return credstring;
                switch(Credential.CredsLevel)
                {
                    case CredsLevels.PrivateID:
                        credstring  = KeyString + Recipient + Domain + License.ToString()↙
     + Credential.ToString();
                        break;
                    case CredsLevels.Email:
                        credstring  = KeyString + Recipient + Domain + License.ToString()↙
     + Credential.ToString();
                        break;
                    case CredsLevels.Identity:
                        Guid userid = (Guid)AdditionalCreds["ID"];
                        string loginname = (string)AdditionalCreds["LoginName"];
                        string DisplayName = (string)AdditionalCreds["DisplayName"];
                        string Password = (string)AdditionalCreds["Password"];
                        string Title = (string)AdditionalCreds["Title"];
                        string FirstName = (string)AdditionalCreds["FirstName"];
                        string MiddleInitial = (string)AdditionalCreds["MiddleInitial"];
                        string LastName = (string)AdditionalCreds["LastName"];
                        string Suffix = (string)AdditionalCreds["Suffix"];
                        string Company = (string)AdditionalCreds["Company"];
                        string CreditCard = (string)AdditionalCreds["CreditCard"];
                        string CreditCardExpiry = ((DateTime)AdditionalCreds["            ↙
    CreditCardExpiry"]).ToString("yyMMdd");

                        credstring  = KeyString + recipient + domain + license.ToString()↙
     + Credential.ToString() + loginname + userid.ToString()
                            + DisplayName + Password + Title + FirstName + MiddleInitial ↙
    + LastName + Suffix + Company
                            + CreditCard + CreditCardExpiry;
                        break;
                    case CredsLevels.Domain:
                        System.Diagnostics.Debug.Assert(false, "No support for Domain IDs↙
     yet");
                        break;
                    case CredsLevels.DomainAndServer:
                        System.Diagnostics.Debug.Assert(false, "No support for           ↙
    DomainAndServer IDs yet");
                        break;
                    case CredsLevels.EmailAndServer:
                        System.Diagnostics.Debug.Assert(false, "No support for           ↙
    EmailAndServer IDs yet");
                        break;
                    default:
                        System.Diagnostics.Debug.Assert(false, "Invalid Creds Level      ↙
    specified for the MICA");
                        break;
                }
                return credstring;
            }
        }

        private string Signature
        {
            get
            {
                if(sigstring != null) return sigstring;
                System.Text.UnicodeEncoding ue = new System.Text.UnicodeEncoding();
                switch(Credential.CryptoType)
                {
                    case CryptoTypes.RSA:
                        sigstring = Convert.ToBase64String(rsa.SignData(ue.GetBytes(     ↙
    CredentialsString), "SHA1"));
                        break;
                    default:
                        throw(new Exception("Invalid Crypto Type specified for the MICA" ↙
    ));
```

```
            }
            return sigstring;
        }
        set
        {
            this.sigstring = value;
        }
    }

    private string KeyString
    {
        get
        {
            if(keystring != null) return keystring;
            switch(Credential.CryptoType)
            {
                case CryptoTypes.RSA:
                    keystring = Convert.ToBase64String(rsaparams.Modulus) + "." +     ↙
Convert.ToBase64String(rsaparams.Exponent);
                    break;
                default:
                    throw(new Exception("Invalid Crypto Type specified for the MICA" ↙
));
            }
            return keystring;
        }
        set
        {
            int n;
            string work = value;
            switch(Credential.CryptoType)
            {
                case CryptoTypes.RSA:
                    n = work.LastIndexOf(".");
                    string exponent = work.Substring(n+1);
                    rsaparams = new RSAParameters();
                    rsaparams.Exponent = Convert.FromBase64String(exponent);
                    work = work.Substring(0, n);
                    string modulus = work;
                    rsaparams.Modulus = Convert.FromBase64String(modulus);
                    rsa.ImportParameters(rsaparams);
                    break;
                default:
                    throw(new Exception("Invalid Crypto Type specified for the MICA" ↙
));
            }
        }
    }

    #endregion Private Methods
    #region Public Methods and Properties
    public bool Verify()
    {
        bool ret = false;
        System.Text.UnicodeEncoding ue = new System.Text.UnicodeEncoding();

        byte[] data = null;
        switch(credential.CryptoType)
        {
            case CryptoTypes.RSA:
                data = ue.GetBytes(CredentialsString);
                ret = rsa.VerifyData(data, "SHA1", Convert.FromBase64String(Signature↙
));

                break;
            default:
                throw(new Exception("Invalid Crypto Type specified for the MICA"));
            }

            return ret;
        }

        public override string ToString()
        {
            if(address != null) return address;

            address = KeyString + "." + Signature + "." + License.ToString() + "." +
    Credential.ToString() + @"@" + Domain;
            address = address.Replace('/', '_');
            address = address.Replace('+', '~');
            return address;
        }

        public Guid ID
        {
            get
            {
                return id;
            }
        }
        public SortedList AdditionalCreds
        {
            get
            {
                return this.additionalCreds;
            }
            set
            {
                additionalCreds = value;
            }
        }
        public string Recipient
        {
            get
            {
                return recipient;
            }
            set
            {
                recipient = value;
            }
        }

        public string Domain
        {
            get
            {
                return domain;
            }
            set
            {
                domain = value;
                this.keystring = null;
            }
        }

        public MICALicense License
        {
            get
            {
                return license;
            }
            set
            {
                license = value;
                this.keystring = null;
            }
        }

        public MICACredential Credential
        {
            get
            {
                return this.credential;
            }
            set
            {
                this.credential = value;
                this.credstring = null;
            }
        }

        public string PrivateKey
        {
            get
            {
                return rsa.ToXmlString(true);
            }
        }

        public string PublicKey
        {
            get
            {
                return rsa.ToXmlString(false);
            }
        }
        #endregion Public Methods and Properties
    }
}
```